Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2013, Article ID 684563, 22 pages
http://dx.doi.org/10.1155/2013/684563

Research Article

Symbiotic Service Composition in Distributed Sensor Networks

Tim De Pauw,[1,2] Bruno Volckaert,[1] Anna Hristoskova,[1] Veerle Ongenae,[2] and Filip De Turck[1]

[1] *Department of Information Technology, Ghent University, iMinds, Gaston Crommenlaan 8/201, 9050 Ghent, Belgium*
[2] *Department of Industrial Technology and Construction, Ghent University, Valentin Vaerwyckweg 1, 9000 Ghent, Belgium*

Correspondence should be addressed to Tim De Pauw; tim.depauw@intec.ugent.be

Received 2 April 2013; Revised 12 October 2013; Accepted 12 October 2013

Academic Editor: Tai-hoon Kim

Copyright © 2013 Tim De Pauw et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

To cope with the evergrowing number of colocated networks and the density they exhibit, we introduce symbiotic networks—networks that intelligently share resources and autonomously adapt to the dynamicity thereof. By allowing the software services provided in such networks to operate in an equally symbiotic manner, new opportunities for the so-called service compositions arise, which take advantage of the multitude of services and combine them to achieve goals set out by the individual networks. To accommodate services in large-scale symbiotic networks, including wireless sensor networks, we propose a software platform which autonomously constructs and orchestrates such compositions. Furthermore, upon changes in the infrastructure, the platform responds by adapting compositions to reflect the changed context. To enable the interaction between services offered by arbitrary partners, the platform deploys ontologies to achieve a common vocabulary and semantic rules to express the policies imposed by the networks involved. By applying the platform to typical scenarios from the field of sensor-augmented cargo transportation and logistics, we illustrate its applicability and, through performance evaluation, show a significant increase in process efficiency. Additionally, by means of a generic problem generator, we quantify the scalability of our platform and show the importance of an appropriate priority function, one of the core constituents of our service composition approach.

1. Introduction

Over the past few years, network environments have seen a vast increase in density. Now more than ever, wired and wireless networks have become truly colocated infrastructures. Today's home and office users have grown to depend on Wi-Fi hotspots as well as 3G and 4G cellular networks and also increasingly rely on wireless sensor networks—based on *ZigBee*, for instance—and interconnected home automation systems.

Despite this explosion in both the number of available network technologies and the number of colocated infrastructures, relatively little progress has been made regarding optimized allocation of resources. Interconnected devices usually contend for the same limited amount of bandwidth rather than efficiently sharing it, leading to all too well-known issues such as network latency and congestion. Figure 1 shows a typical office environment, in which several wireless networks coexist with a wired network as well as a public cellular network. Despite the scarcity of their resources, the networks make no effort to cooperate. Consequently, typical issues such as signal interference and lack of bandwidth occur frequently. In extreme cases, PCs and smartphones will lose connection, smart TVs will not be able to deliver rich content, climate control will malfunction, and telephone calls will be dropped. And yet, most of these networks share the same fundamental technologies and should therefore be able to cooperate intelligently!

Inspired by symbiotic organisms encountered in nature, our goal is to mitigate such problems through the introduction of *symbiotic networks*. We define a symbiotic network as an environment that originates when two or more networks engage in mutually beneficial interoperation. This is achieved by sharing resources such as bandwidth and computational power between the symbiotic partners, crossing layers and boundaries in the process. We aim for network communication across layers and logical and physical boundaries in a technology-agnostic fashion. This results in more robust,

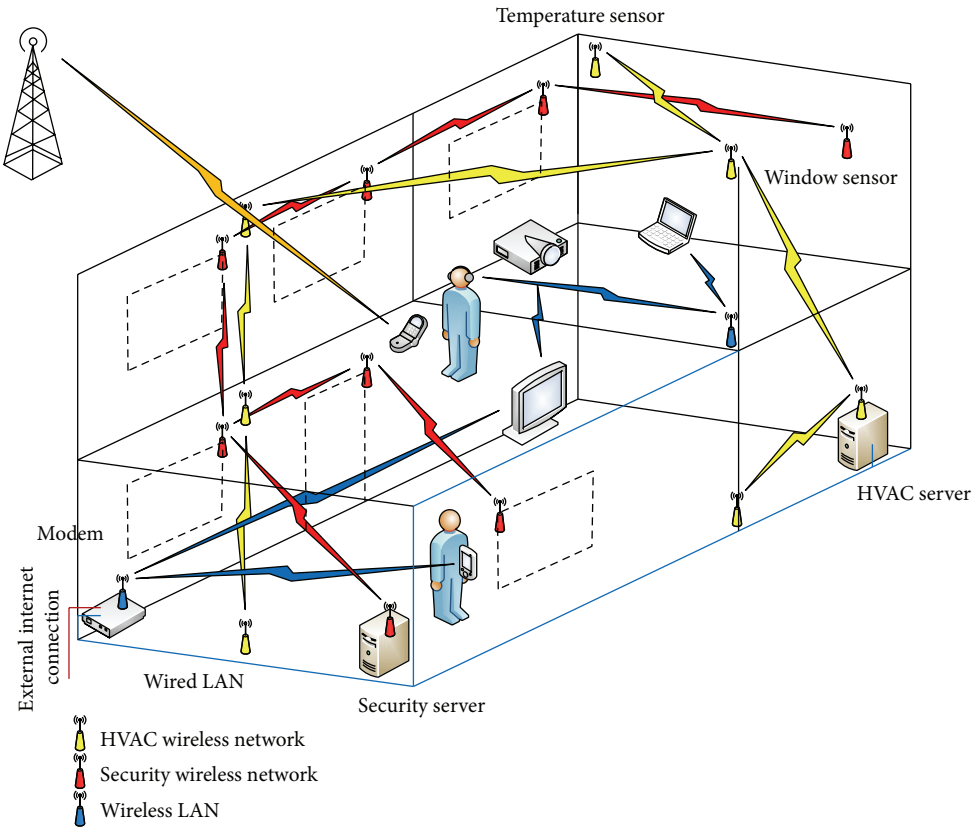


Figure 1: A typical office environment, with no sharing of network resources.

dependable, and scalable networks and increases the performance and energy efficiency of the environment as a whole. Applied to the office environment described above, we would get the result displayed in Figure 2.

Symbiotic networks may scale infinitely. Because of their dynamic nature, constituent networks may arrive and depart at any time as may the individual nodes that form them. To cope with the scale of the infrastructure on one hand and its frequently changing nature on the other hand, the symbiotic network must be able to operate autonomously. Furthermore, mechanisms need to be in place to govern the interaction of partners that may or may not be aware of each other's existence, in order to deal with concerns in terms of confidentiality, reliability, timeliness, an so on. A large amount of research has already been carried out demonstrating the feasibility and advantages of such an approach [1, 2].

Modern-day network environments already make heavy use of each other's *services*, for instance, to outsource features to specialized partners. Expanding on this principle, our symbiotic service platform intelligently combines functionality provided by symbiotically interacting partners in order to meet set goals. Symbiotically interoperating network environments may each offer a set of *software services*, which their symbiotic partners may utilize to achieve their compound goals. They allow the partners to mix and match service invocations from various networks. The symbiotic environment must therefore be aware of which networks offer which services, how their concepts relate to one another, how the services can be invoked, and which constraints are imposed upon this process. Furthermore, because of the dynamic nature of the symbiotic network, the set of available services also changes over time—either slowly or fast, depending on the nature of the network.

We therefore introduce a symbiotic service platform, which provides a common infrastructure for symbiotically interoperating networks to employ each other's services. Individual goals are met by constructing *service compositions*: graphs that model the invocation of available services to produce the information necessary to ultimately achieve the specified goal. Relying on Semantic Web technology, we define a robust yet flexible model for sharing vocabularies and taxonomies between symbiotic partners. Employing this model, our tunable best first search algorithm *SeCoA* produces service compositions to meet a given goal. These compositions are then translated to software deployment schemes and executed on the symbiotically enabled network devices. By subsequently monitoring the operation of the deployed software, feedback is collected to ensure the continued operation of the composition. If, for any reason, the goal can no longer be satisfied or if an opportunity for a better composition arises, the platform adapts the composition accordingly.

To evaluate the symbiotic service platform, we applied it to typical problems from the field of cargo transportation

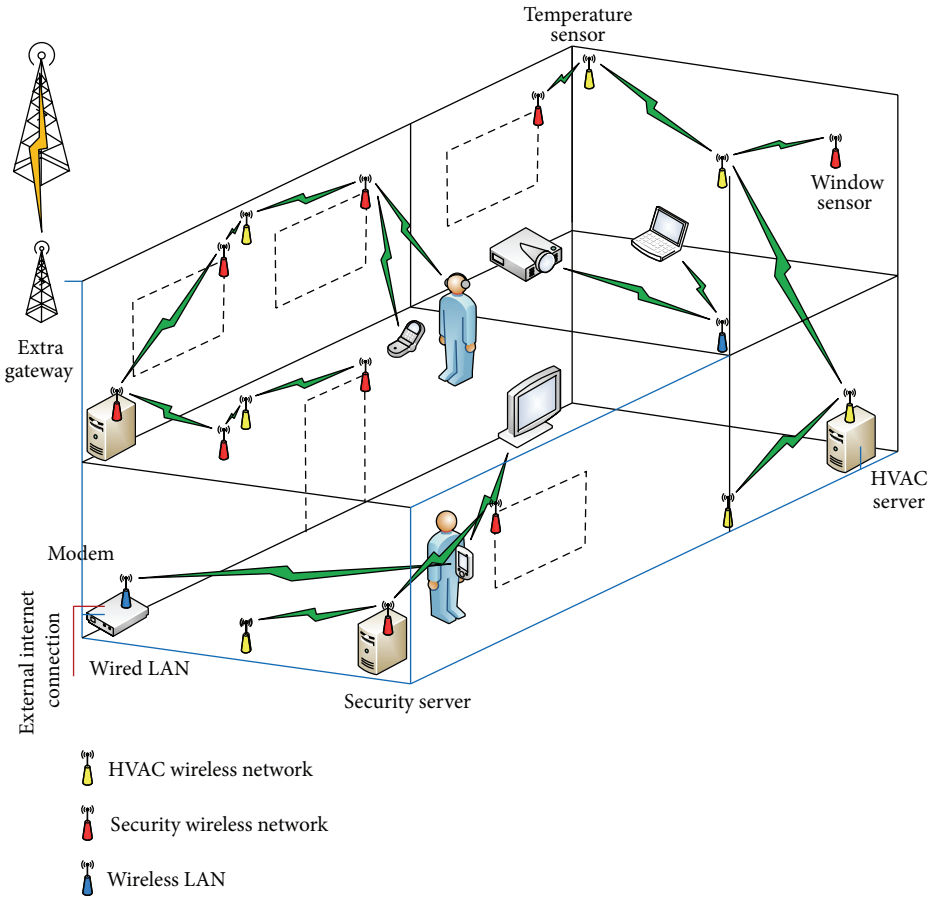


Figure 2: The same office environment, but with symbiotic networking capabilities.

and logistics. In this domain, many partners continuously contact each other's systems to align and realize goals such as verifying cargo manifests or ensuring proper treatment of certain items, based on measurements from wireless sensors. In case of unforeseen circumstances, additional partners are often added to the mix; for instance, local authorities might need to be contacted in case of a health hazard. Geographically speaking, the problem's domain is vast: cargo might be transported anywhere in the world and consequently be subject to highly complex local restrictions. Keeping track of these countless, often changing rules requires a significant amount of often manual effort from international logistics operators. Thus, there is a clear need for unambiguous modeling of the problem domain and the mechanisms it is comprised of. Using Semantic Web technology, we are able to capture such information unambiguously and use it to detect opportunities for autonomous goal realization through service composition.

To mimic day-to-day operation in a cargo transportation context and replicate typical goals that occur in it, we developed a logistics simulator. Applying our service platform to this context, discovered goals are autonomously solved by composing and enacting symbiotic services made available by the symbiotically enabled partners. Through benchmarks of the service platform, we will show that our solution can obsolete a great deal of manual effort, reduce and even eliminate interpretation errors, and highly increase cargo throughput.

Our simulations inspired by the logistics domain primarily focus on the practical applicability of symbiotic service composition. To examine our approach in terms of scalability, we also developed a generic problem generator, which we used to further benchmark our algorithm.

The remainder of this paper is structured as follows. In Section 2, we provide a formal statement of the service composition problem in symbiotic networks. Section 3 describes the platform we propose to tackle the problem, along with the service model that underpins it. In addition, we describe our symbiotic service composition algorithm SeCoA and conduct an analysis of it. In Section 4, we apply the problem to real-life scenarios from the field of cargo transportation and logistics. We elaborate on the simulator which we developed to enact these scenarios for evaluation purposes and discuss the results we obtained from it. Section 5 describes our generic problem generator, which is subsequently used to assess the scalability of SeCoA. After taking a look at related work, we end this paper with our conclusions and opportunities for future research.

2. Problem Statement

Evidently, our symbiotic service platform must accommodate software services. In this section, we describe the model we employ to describe such services and include them in compositions.

We assume that each service is accompanied by two lists: one containing *input parameters*, and one containing *output parameters*. While both may be empty, composition is parameter based: given an optional set of *initial services*, a *goal service*, and a set of other available *services*, the objective is to create a directed acyclic graph $G = (V, E)$ such that

(1) each element of V corresponds to a service,

(2) each element of E represents the exchange of a parameter value between two services,

(3) there is at least one path from all the vertices corresponding to the initial services to that corresponding to the goal service,

(4) the value of each input parameter of each service included in the composition is provided by that of exactly one output parameter of another service.

Each parameter has a *type*, given by a class name. Classes may optionally be equivalent or inherit from one another. If service A produces an output parameter x of type X and service B requires an input parameter y of type Y, then the value of x may be used as the value of y if and only if X is either equivalent to Y or a subclass of it.

Note that while only a single goal service may be provided, one can easily extend the model to multiple goals through the introduction of a metaservice, which takes the output parameters of the actual goal services as its input parameters.

An important aspect of symbiotic networks is that services in a composition may belong to different parties, who agree upon a common vocabulary. To further govern the interoperation of these parties' services, we introduce so-called *policies*. A policy is of the form *antecedent* ⇒*consequent*. If the condition expressed by the antecedent is met by an edge $e \in E$, then the consequent is applied to the services represented by the vertices which e connects.

An antecedent checks if the input parameter, output parameter, and the services, respectively, providing and consuming them meet certain requirements. A trivial antecedent might state that the service consuming the parameter is located in a particular network taking part in symbiotic interoperation. However, far more complex logic can be employed, as we assume antecedents to be freeform rules. They apply to properties of the edge e, the service vertices which it connects, and the environment in which they exist.

The application of a consequent consists of the addition of *filters* to the parameter exchange. A filter may be applied to the service providing the parameter, the service consuming it, and the parameter itself. We further distinguish between the application of a parameter filter when the providing service emits it and when the consuming service receives it. Thus, there are four distinct types of filters. These filters are an abstract representation of nonfunctional requirements.

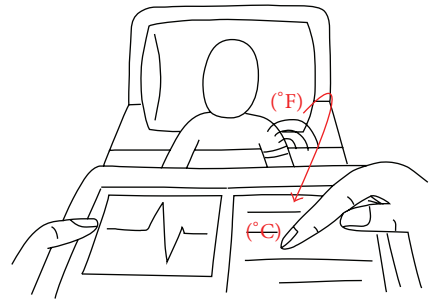


Figure 3: A healthcare professional inspects a sensor temperature reading, obtained from a patient's body area network and subsequently converted to a different temperature scale.

Examples include a parameter filter stating that a particular value must be transmitted securely or that it must be measured with a given accuracy. A service filter might impose that the service operates with elevated permissions.

The graphs that are constructed based on this service model represent invocations of symbiotic services, where information is passed between two services at each invocation. Invocations occur either serially or in parallel, depending on the structure of the graph.

As an example, consider a simple scenario from the field of healthcare, informally visualized in Figure 3. To monitor a patient's vital signs, wireless sensor devices are placed on his body, forming a so-called *wireless body area network* [3]. One of the measurements that these sensors emit is the patient's body temperature; this happens by means of a *TemperatureProvider* service. This service emits one output parameter, namely, a *FahrenheitTemperature*. Evidently, the hospital wishes to consume this temperature value and runs a *TemperatureConsumer* service on its equipment. Unfortunately, this service was developed elsewhere and only accepts a *CelsiusTemperature*. Thus, given *TemperatureProvider* as our sole initial service and *TemperatureConsumer* as our goal service we cannot construct a composition. Fortunately, the hospital also exposes a *TemperatureConverter* service that takes an input *FahrenheitTemperature* and outputs a *CelsiusTemperature*. Wiring up the three services, we would obtain a trivial composition that satisfies our goal.

As it stands, communication between the patient's wireless body area network and the hospital's network is transmitted without any security measures. Since we are dealing with personal information, the hospital might be wise to introduce some confidentiality. It does so by adding a policy, which states that any parameter exchange between the patient's network and its own mandates the application of two filters. The first is an *Encryption* filter, applied to the output parameter of the providing service; the second is a *Decryption* filter on the input parameter of the consuming service.

Another example might be a policy stating that all readings obtained from wireless sensors must be sufficiently accurate. Such a policy's antecedent would be "if the service providing the parameter is a sensor service," and its consequent would be "apply an *Accuracy* filter to the providing service." The same effect can be obtained using a policy

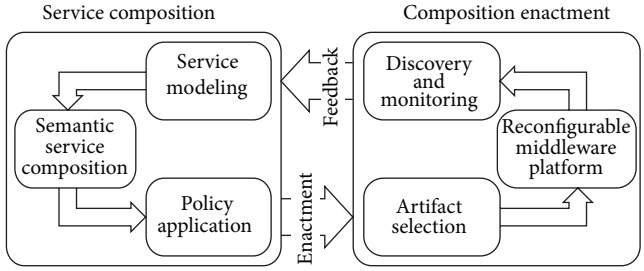


FIGURE 4: High-level architecture of the symbiotic service composition platform.

which checks if the consuming service is a sensor-consuming service. One can also apply multiple policies at will.

In what follows, we will use this simple scenario to clarify the functionality of our service platform and its constituents. For the sake of simplicity, we will only consider the confidentiality policy; the other two policies that we have described are largely similar.

3. Proposed Solution

3.1. Platform Architecture. The architecture of the software platform that we propose for the construction, enactment, and life cycle management of symbiotic service compositions is visualized in Figure 4. At the highest level, it consists of two main components.

The first component, *Service Composition*, deals with the descriptions of individual software services provided by symbiotic networks and uses them to construct compositions. The *Service Modeling* component employs Semantic Web technology to formally describe the available services and their context. This information is subsequently used to construct compositions that satisfy certain goals. As explained, policies are added to these compositions to represent nonfunctional requirements pertaining to the symbiotic environment.

The produced high-level service compositions are passed to the second component, entitled *Composition Enactment*. This component deals with the infrastructural needs of the platform. It translates a composition to a set of interconnected software artifacts, deploys them on the resources at hand, and ensures their proper operation. Vital information pertaining to the infrastructure is passed back to the *Service Composition* component, giving rise to a feedback loop.

In this paper, we mostly focus on the *Service Composition* component. It should, however, be noted that our platform architecture is mostly technology agnostic. The compositions produced by our algorithm *SeCoA*, discussed in the next sections, can be enacted on any middleware platform, as long as that platform is capable of supporting symbiotic services. Conversely, symbiotic middleware is not restricted to the use of the SeCoA algorithm.

3.2. Domain and Service Model. To describe symbiotic services in an expressive fashion, we rely upon Semantic Web technology. We employ the *OWL-S* ontology [4] to create semantic service descriptions. OWL-S builds upon *OWL*, the *Web Ontology Language* [5], which in its turn extends *RDF*, the *Resource Definition Framework*.

Using formal semantics, OWL ontologies allow for the consistent description of concepts and the relationships between them. Such ontologies are comprised of *classes*—the root class being *owl:Thing*—and instances thereof, called *individuals*. Between these individuals, relationships can be defined by instantiating *properties*. *Datatype properties* couple individuals to RDF literals or XML schema datatypes, whereas *object properties* define relationships between individuals.

OWL-S is an OWL ontology that is used to describe Semantic Web Services. By interpreting the high-level descriptions of services, applications can autonomously discover and invoke them. These descriptions consist of three main parts.

(i) The *service profile* provides basic information about what the service does. While it is mostly comprised of human-readable metadata, the service profile may also be used for the purpose of service discovery.

(ii) In the *process model*, the service's so-called *IOPE's* are expressed: inputs, outputs, preconditions, and effects. They determine the parameters provided to and produced by the service, as well as the conditions under which the service can be invoked and those resulting from its invocation. Conditions are expressed as rules, traditionally in *SWRL*, the Semantic Web *Rule Language* [6].

(iii) Finally, the *service grounding* details how the service can be invoked. It maps inputs and outputs to protocol-specific parameters, for example, via WSDL.

OWL-S's process model is fairly elaborate. By means of control constructs such as *Sequence*, *If-Then-Else*, and *Repeat-While*, highly complex process flows may be modeled. Nevertheless, only services with a single-step process are currently considered by our platform. They can either be described by *atomic* or *simple processes*. The former are concrete representations of single-step operations, whereas the latter provide the means to abstract any process.

The resulting composition itself, on the other hand, can be a candidate for such abstractions, as it is modeled as a *composite process*. Using the *Sequence* and *Split-Join* control constructs, the underlying atomic and simple processes are combined to form the aforementioned graph G.

While the OWL-S ontology is targeted at Web Services, it is mostly applicable to various domains. We therefore believe that it is very suitable to heterogeneous environments with a more extensive service model, such as symbiotic networks.

The OWL-S standard provides the foundations to semantically describe and operate services, but, naturally, it does not explicitly provide a framework to do so in symbiotic networks. Therefore, we introduce a set of extensions to the ontology, which allow us to express symbiotic-network-specific concepts.

Table 1: OWL classes accommodating symbiotic service composition.

Class	Superclass
Policy	owl:Thing
Antecedent	expr:Expression
Consequent	owl:Thing
ParameterFilter	owl:Thing
ServiceFilter	owl:Thing
PolicyList	rdf:List

Table 2: OWL properties accommodating symbiotic service composition.

Domain	Property	Range
Policy	hasAntecedent	Antecedent
Policy	hasConsequent	Consequent
Consequent	appliesOutputFilter	ParameterFilter
Consequent	appliesInputFilter	ParameterFilter
Consequent	appliesProviderFilter	ServiceFilter
Consequent	appliesConsumerFilter	ServiceFilter

In Section 2, we introduced policies and filters. These translate to the OWL classes and properties shown in Tables 1 and 2. Two of them require additional clarification.

(i) The individuals in OWL ontologies are not ordered. However, policies take precedence over one another. Consequently, the class *PolicyList* is introduced to impose an order.

(ii) A policy's *Antecedent* is a SWRL *Expression*, much like the preconditions and effects defined in an OWL-S process. Antecedents may employ the SWRL variables *Producer* and *Consumer* to refer to the two services involved in the exchange, and the variables Output and Input to refer to the parameter values.

If we apply these concepts to the patient care scenario that we introduced earlier, we obtain the OWL-S services shown in Table 3. In addition, we introduce the OWL class *AdministrativeDomain*, with its two instances *Patient* and *Hospital*, to represent the two symbiotically interoperating environments involved in the scenario. A new OWL object property *hasAdminDomain* links services to such domains. Harvesting these ontology additions, we can use SWRL to express our confidentiality policy semantically:

> *hasAdminDomain*(*Producer, Patient*) ∧ *hasAdminDomain*(*Consumer, Hospital*) ⇒ apply *Encryption* to *Output* and apply *Decryption* to *Input*

Thus, for each parameter exchange in the constructed service invocation graph, we will check if the service providing the parameter is in the administrative domain called *Patient* and its consuming counterpart is in the *Hospital* domain. If so, the parameter provided by the former will receive an *Encryption* policy, and before the latter consumes its value, a corresponding *Decryption* policy will ensure that the service can interpret the parameter's value.

Figure 5 shows a summary of the symbiotic service composition which we just modeled using Semantic Web technology. In this mostly trivial case, we were able to wire up the services manually. Of course, in realistic environments, composition problems will often be far less straightforward and/or far more numerous. Therefore, in the next section, we discuss our approach toward automated service composition in symbiotic networks.

3.3. Service Composition Algorithm. Using the semantic service descriptions just outlined, we are able to deduce if services are capable of interacting with one another and construct the desired compositions that realize given goals. The algorithm we devised for doing so is called *SeCoA* and consists of three phases.

(1) Weed out Unsatisfiable Inputs. In this initial preprocessing phase, SeCoA eliminates services which require input parameters that are not provided by any of their counterparts, as these can never take part in a composition. If neither the type of an input nor a subtype of it can be provided, the service requiring that particular type will not be considered any further.

Phase 1 is optional; in symbiotic networks where the interoperating service environments are well tuned to one another, this additional analysis phase might slow down the composition process. Conversely, highly dynamic symbiotic networks might consist of environments that are completely unaware of each other's concepts and services, resulting in a large reduction of the set of candidate services for the next phase and greatly improved processing time.

(2) Match Outputs to Inputs. The best first search algorithm for constructing a composition is shown in Listing 1; it was partly inspired by the *WTE+* algorithm [7], which was also developed at iMinds. Starting from the goal service, it attempts to work its way up to the initial services (if any) by finding services that provide unsatisfied input parameters. Thus, initially, the solution graph only contains the goal service, and its input parameters are to be found.

In the first iteration, one of these unknown inputs is examined. For each service which provides an output parameter of the same class or a subclass, a queue entry is created. Subsequent iterations gradually satisfy the remaining inputs until the first valid solution is found; that is, it attempts to find a composition which satisfies all the input parameters encountered and which contains all of the given initial services. If a service is already part of the composition, the corresponding graph vertex is reused. Additional verification is performed to make sure that the resulting composition does not contain any cycles; this is exemplified by the call to the *hasAncestor*() function.

The order in which subsequent (incomplete) candidate compositions are selected for expansion depends on their priority, given by

$$p(c) = \alpha \times i_c + \beta \times u_c + \gamma \times s_c, \quad (1)$$

where c is the partial composition being examined, i_c is the number of initial services not included in c, u_c is the number

Table 3: OWL-S services involved in example healthcare scenario.

Service	Input type	Output type
TemperatureProvider	—	FahrenheitTemperature
TemperatureConsumer	CelsiusTemperature	—
TemperatureConverter	FahrenheitTemperature	CelsiusTemperature

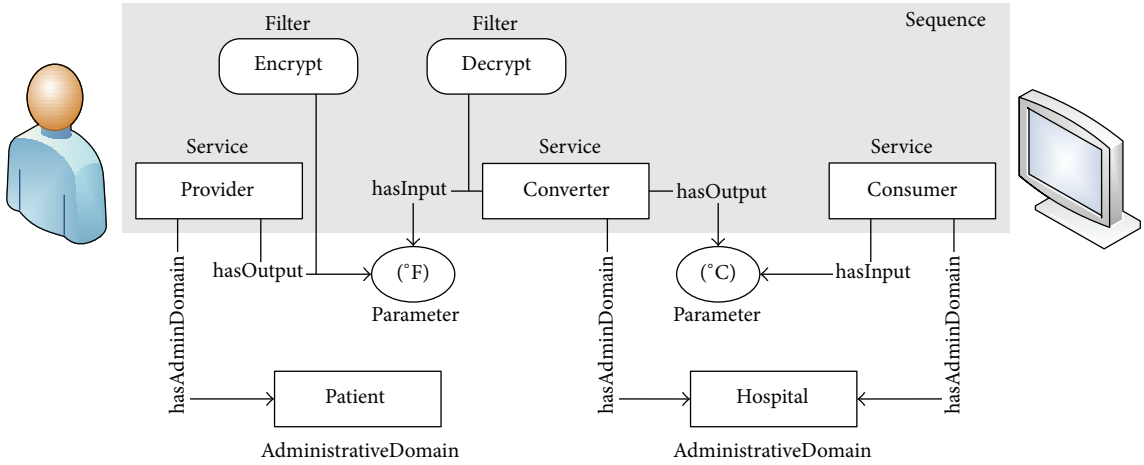


Figure 5: Composition ontology, for example, healthcare scenario (simplified).

```
composition ← Graph()
queue ← PriorityQueue()
addVertex(composition, goalService)
push(queue, {composition, inputs(goalService)})
while queue not empty do
    {composition, remainingInputs} ← pop(queue)
    input ← pop(remainingInputs)
    for provider ∈ servicesProviding(type(input)) do
        if hasAncestor(provider, service(input)) then
            continue
        newComposition ← composition
        newRemainingInputs ← remainingInputs
        if provider ∉ newComposition then
            addVertex(newComposition, provider)
            push(newRemainingInputs, inputs(provider))
        addEdge(newComposition, provider → service(input))
        if newRemainingInputs empty then
            if initialServices ⊂ newComposition then
                return newComposition
        else
            push(queue, {newComposition, newRemainingInputs})
```

Listing 1: Phase 2 of SeCoA: construction of the service composition graph.

of input parameters in c which are still unsatisfied, s_c is the total number of services involved in c, and α, β, and γ are tunable weight coefficients.

This function can be further customized at will. For instance, in previous research, we have also taken into account the number of symbiotic partners in c [8].

(3) Apply Policies. Provided that phase 2 of the algorithm produced a valid composition, the list of policies is exhaustively checked against each of its parameter exchanges. If an antecedent match is encountered, the filters associated with the policy's consequent are applied to the services and parameter values involved.

SeCoA focuses on constructing service compositions rather than adapting them to dynamic changes in the ontology. Nevertheless, the performance evaluation we describe in Section 4 shows that the algorithm is also suited for use in dynamic environments.

3.4. Example Algorithm Application. To clarify the SeCoA algorithm, we return to our running example once more and construct the composition shown in Figure 5 programmatically. Let us assume that the hospital does not deem phase 1 of the algorithm necessary, because most if not all of its services' input parameters can be satisfied. Thus, we will skip ahead to phase 2, in which the actual composition is constructed.

As mentioned, the goal service is *TemperatureConsumer*; it has one input parameter of the type *CelsiusTemperature*. There is a single initial service *TemperatureProvider*, which outputs a *FahrenheitTemperature*. Furthermore, the hospital provides a *FahrenheitTemperatureConverter* and a *KelvinTemperatureConverter*, which, respectively, take a *FahrenheitTemperature* and a *KelvinTemperature* and turn those into a *CelsiusTemperature*.

Initially, the queue contains a single item: a composition consisting of solely the goal service *TemperatureConsumer*, accompanied by its only input parameter *CelsiusTemperature*. In the first iteration, the algorithm looks for services providing parameters of a compatible type. Two of these services are found, namely, the two *TemperatureConverter* services. For each of those, a new composition is constructed by prepending them to the *TemperatureConsumer* service. Because neither of those compositions contains the initial service and because they each introduce one unsatisfied input parameter, the algorithm proceeds. It does so by adding two queue items—one for each *TemperatureConverter* service's input parameter.

In the second iteration, there are two items on the queue. Depending on the implementation of the priority function $p(c)$, either the one for *FahrenheitTemperatureConverter* or the one for *KelvinTemperatureConverter* will be processed first. In our default implementation, both partial compositions would yield the same value for $p(c)$. Let us therefore be pessimistic and assume that *KelvinTemperature* comes first. In this case, the algorithm would fail to find an input parameter of the type *KelvinTemperature*. In fact, had we not skipped phase 1 of the algorithm, then the service would have been eliminated already!

Table 4: Variables used in complexity analysis of SeCoA.

Symbol	Description
s	Total number of candidate services considered for composition
$P(R)$	Probability of service reuse at a given iteration of SeCoA phase 2
t	Average number of services which provide a given parameter type
j	Average number of input parameters per service
p	Total number of policies enforced in SeCoA phase 3

Proceeding to the third iteration, the algorithm now considers *FahrenheitTemperatureConverter* and prepends it to the composition. The queue consequently contains a single item, corresponding to a composition lacking only a *FahrenheitTemperature*. Therefore, the fourth iteration is the final one; the algorithm adds *TemperatureProvider*, decides that all input parameters are satisfied and that all initial services are present, and produces the desired result.

We assume that phase 3 of the algorithm, in which policies are applied to parameter exchanges, is sufficiently clear. Thus, for the remainder of this paper, we will move on to a more extensive problem context, used to evaluate our symbiotic service composition platform.

3.5. Algorithm Complexity Analysis. In this section, we provide an analytical characterization of SeCoA's performance. By examining the time and space complexity of the algorithm's three phases, we obtain an expression for its overall worst-case performance. The variables used in this analysis are summarized in Table 4.

Phase 1 of SeCoA examines every input parameter of each service provided exactly once. Thus, for a total of s services, which have an average of j inputs per service, the time complexity of the first phase amounts to $O(j \times s)$, in the best, average, and worst case.

In phase 2, we employ the best-first search heuristic. Provided that one selects an optimal priority function $p(c)$, the optimal path through the search tree can be followed straightaway. Thus, the best-case time complexity is linear in the number of services included in the solution. To obtain phase 2's worst-case time complexity, let us consider the case where the full search tree is traversed; thus, either the worst possible priority function has been specified or no suitable composition can be constructed. Processing a candidate solution involves examining the next unresolved input parameter in the composition and adding an entry to the priority queue. As there are on average j input parameters to each service and t services that can provide each type of parameter, the number of newly added tree nodes would average out to $j \times t$. However, this expression does not yet account for service reuse. As discussed in Section 3.3, SeCoA reuses the output of services which are already part of the composition. We define $P(R)$ as the probability of being able to reuse a service instead of introducing a new one into the composition. Then, an average of $j \times t \times (1 - P(R))$ search

tree nodes is added per iteration. This corresponds to the notion of an *average branching factor*, often used in data structure analysis. For a tree with an average branching factor b, a best-first tree search algorithm has a worst-case time complexity of $O(b^d)$, where d is the depth of the tree [9]. In the extreme case that all s provided services are part of the optimal composition, the tree depth d potentially rises to $j \times s$. Thus, the worst-case time complexity of phase 2 of our algorithm is $O((j \times t \times (1 - P(R)))^{j \times s})$.

Phase 3 examines each edge of the produced composition against each policy antecedent. Thus, in the extreme case, applying p policies to a composition containing all s services will involve a worst-case time complexity of $O(p \times s)$.

Consequently, for all three phases combined, the SeCoA algorithm has a total worst-case *time* complexity of $O((j + p) \times s + (j \times t \times (1 - P(R)))^{j \times s})$. However, in Section 4, we will show that the algorithm performs well for real-life scenarios. Firstly, this is due to the fact that SeCoA's *average-case* performance, while being difficult to quantify exactly, is intuitively a lot better than that in the worst case. Secondly, phase 1 of the algorithm reduces the number of services considered in the second phase, effectively lowering the factor s in the exponent $j \times s$.

Regarding *space* complexity, only phase 2 of SeCoA employs additional data structures and therefore requires investigation. We already know that each queue entry consists of a partial composition and a list of input parameters to resolve. Each partial composition is a graph consisting of up to s vertices with up to $s \times 1/2(s - 1)$ edges between them in the case of a complete graph. In the worst case, the list of input parameters contains $j \times s$ elements. Adding up both components, the space complexity of a single queue entry becomes $O((s + j) \times s)$.

The queue as a whole can contain as many entries as the entire search space holds; we have already determined the size of the search tree to be $O((j \times t \times (1 - P(R)))^{j \times s})$. Thus, we obtain a worst-case space complexity of $O((j \times t \times (1 - P(R)))^{j \times s} \times (s + j) \times s)$ for phase 2 of SeCoA.

4. Cargo Transportation and Logistics Simulation

4.1. Illustrative Scenarios. One particular domain that would greatly benefit from an application of symbiotic networks, with greater emphasis on the problem scale, is that of cargo transportation and logistics. In this field, vehicles transport goods between locations, crossing borders in the process. Many networks are involved: logistics operators, warehouses, customs offices, local authorities, insurance agencies, and emergency services, and of course vehicles, containers, and crates may be equipped with various networked devices, ranging from back-end servers over tablet computers to wireless sensors and actuators. In the near future, drivers themselves may even carry unobtrusive sensor devices to monitor their vital signs.

Having all these network environments engaged in symbiotic interoperation not only allows them to communicate in a more reliable fashion but also paves the way for symbiotic service composition. After all, many circumstances arise where near-instant service composition can streamline the numerous and often error-prone processes that occur day to day. Apart from reducing errors, this would also result in faster response to changing circumstances and allow for fully automated and autonomous business processes.

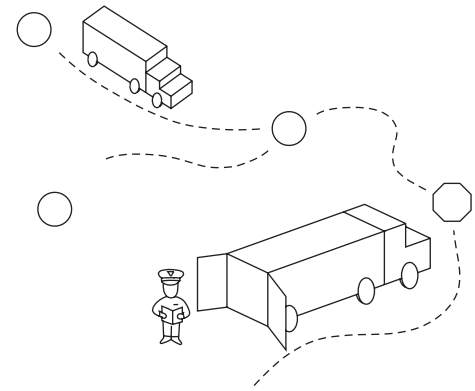


Figure 6: A government official inspects legal documents, delaying the vehicle's cargo. At the same time, many other vehicles (of which one is shown) are also en route.

Consider an example where a vehicle is waiting in line at a country border. The cargo is being delayed and the precious time of the driver and the logistics operators, among others, is being wasted. Eventually, the driver hands the required documents to an official, who checks them manually, as shown in Figure 6. If the border officials are thorough, they will also want to inspect every piece of cargo individually. In case of a violation, or merely doubt thereof, the proper authorities need to be contacted. Telephone calls are placed, bureaucracy is set in motion, and the cargo is delayed even further.

But what if this entire process could be automated? Indeed, a software service running on the vehicle could be aware of the cargo and a service at the border office could receive this information to yet another service that instantaneously verifies the validity of the electronic shipping manifest. Moreover, if anything is out of order, services put in place by the authorities could be informed automatically. Symbiotic interoperation between these parties would pave the way for such scenarios. Furthermore, by reducing the net cost of validation processes, rigorous inspection rather than arbitrary sampling might become a valid alternative, which would ultimately result in safer, more trustworthy logistics operation.

Of course, such a scenario introduces a fair amount of challenges. There is the matter of authenticity verification: how can an automated service verify that a cargo item has not been tampered with? Another issue is that of confidentiality, where secure, authentication-based communication patterns need to be established, so that other users of the infrastructure—symbiotic or otherwise—are only granted access to the information they require; for instance, a vehicle often carries cargo provided by various owners, but those

owners do not need to be aware of the contents of each other's shipments. Additionally, what if a network path's failure prevents a crucial message from going through? One final example is traceability: government officials might need quick access to the path which a shipment has followed, for instance, in case of contaminated goods.

This one example is by far not the only scenario from the field of transportation and logistics where symbiotic interoperation could offer a vast improvement. To further illustrate the applicability of our solution, consider the following additional opportunities for symbiotic service composition.

(a) Before a cargo item is loaded onto a vehicle, the vehicle's manifest is inspected to ensure that it does not contain any items that are incompatible with the new one. For instance, a vehicle might be prohibited from simultaneously transporting food items and poisonous goods to avoid contamination.

(b) As an additional security measure, we can extend this measure over time: if a vehicle has recently transported poisonous goods, it is prohibited from transporting food items.

(c) Upon arrival at border security, a vehicle's shipping manifest is checked for prohibited goods. If local law disallows certain items, the logistics operator is informed.

(d) When a vehicle arrives at a warehouse, temperature readings from the wireless sensors installed on cargo items are processed to ensure that they did not exceed predefined thresholds. Other examples of sensor readings include the humidity of food items and data from pressure sensors which detect excessive shocks.

(e) If the thresholds have been exceeded, an alert is sent to the logistics operator and the owner of the cargo items. If any health hazard has been or may have been created, the proper authorities are alerted.

(f) While a vehicle is in transit, readings from the wireless sensors mounted on cargo items are periodically inspected to ensure that they are within accepted ranges. If this is not the case, the vehicle's network might be informed to attempt to rectify the situation, perhaps by adjusting climate control. If the driver needs to intervene, he is alerted as well. In case of dangerous conditions, emergency services and authorities are alerted.

(g) While a vehicle is in transit, readings from the driver's body area network are periodically inspected to ensure that he is in good health. If anything out of the ordinary is found, diagnostic services recommend a remedy. This might, for instance, entail getting some rest or obtaining over-the-counter pharmaceuticals. In a graver situation, such services might also calculate directions to the nearest hospital.

All of these examples are subject to additional complexities, which are generally seen in various types of network environments—be they symbiotic or not. We chose the logistics problem domain for our research because it clearly exhibits such complexities.

Firstly, we assume that the necessary communication mechanisms exist between the symbiotically operating networks. In what follows, we assume that a functional symbiotic infrastructure has been formed at the network level—as described in [2]—and that the necessary endpoints are available for services to connect to one another. This does not imply that all the services involved are based on the same technology stack; after all, one might be dealing very well with devices ranging from resource-constrained sensor nodes all the way up to highly powerful back end servers, each with their own programming languages, libraries, and APIs. Using service description and discovery mechanisms provided by the OWL-S ontology, one can align their interfaces to one another.

Secondly, regulations are traditionally stated in the language of the country or region where they are applied, introducing a need for translation mechanisms. This concerns not only core domain concepts but also units of temperature, pressure, weight, price, and so forth. Our approach based on Semantic Web technology allows us to model such concepts in a highly expressive fashion.

Thirdly, because of the large number of symbiotically interoperating networks—belonging to vehicles, cargo items, warehouses, customs offices, logistics operators, cargo owners, and even drivers—additional mechanisms need to be in place to ensure that the communication between them respects their individual requirements. For instance, the symbiotic service environment must ensure that cargo owners cannot obtain access to each other's shipment information, by encrypting traffic originating from the vehicle and the sensors mounted on the cargo. Again building upon the Semantic Web, our previously described SWRL-based policies provide a human- and machine-readable language for expressing such constraints.

Lastly, the problem domain has a highly dynamic nature. Apart from evidently changing vehicle inventories, among others, local regulations have been known to change suddenly as well, for instance, in the event of a public health crisis. Oftentimes, this will not be properly communicated to logistics operators and warehouses, let alone drivers. This can lead to tedious inspection procedures and rerouting of cargo, which is of course not desirable. By agreeing on a common basic vocabulary—OWL in our case—one can greatly reduce the impact of these contextual changes. Any change in the legislation-related ontologies is immediately picked up by the platform and reflected in the compositions generated by it.

While the introduction of Semantic Web technology does add a degree of complexity to the problem space, we are confident that real-life scenarios would greatly benefit from it. In terms of user-friendliness, the actors involved can depend on a knowledge modeling and acquisition tool such as *Protégé* [10]. Alternatively, the ontology-related terminology could be obscured by a more entry-level modeling tool. Finally, it should be noted that while our platform heavily relies on Semantic Web technology and we employ it to our advantage, none of the concepts that we introduce in this paper actually mandate its use.

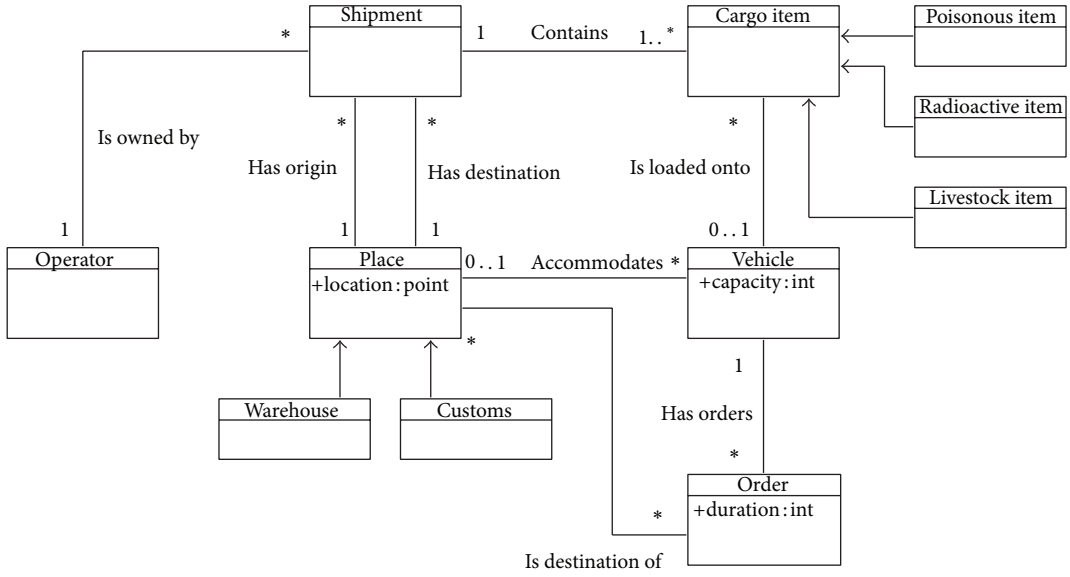


Figure 7: UML class diagram showing the domain model for the developed logistics simulator.

4.2. Simulator Model. Out of the example scenarios described above, we selected a, d, and e for an evaluation of our symbiotic service platform, as they are the most challenging with regard to the complexities described. We implemented a logistics simulator, which mimics a context in which the scenarios naturally occur as goals. These goals are then addressed by our platform.

It is important to note that we purposefully opted for a simplified model of the transportation and logistics context, as our goal is not to accurately simulate the transport operations themselves. Rather, we wish to illustrate and evaluate the applicability of symbiotic services to support common tasks in such scenarios. Thus, although many researchers have applied themselves to problems like route planning and bin packing, they are largely beyond the scope of this paper.

Figure 7 shows the domain model of the simulator that we implemented. Each *Vehicle* is loaded with *Cargo Item*s, with each belonging to a single *Shipment*. Such a shipment is the property of a logistics *Operator* and needs to be transported between two *Place*s. *Vehicle*s have an itinerary, consisting of *Order*s, which dictate which *Place*s a vehicle is to visit during the course of the simulation; the (anticipated) travel *duration* is calculated prior to the actual simulation, whereas the time spent at each place will vary depending on the invocation of the appropriate services. In addition, a vehicle has a *capacity*, which is the number of cargo items it can carry. We refrain from having this capacity depend on properties of the cargo or the vehicle, so as not to overcomplicate the scenarios.

Each simulation consists of the following steps.

(1) *Map Generation.* A weighted directed graph is generated to simulate a geographical setting. The graph's nodes represent places which vehicles will frequent, whereas the edges that correspond to the time required to travel between the two nodes which they connect. This time is expressed as a natural number and is assumed to be fixed. Places are assigned a uniformly distributed latitude and longitude, and the travel duration between two places is given by a fixed minimum (viz., the Euclidean distance between both places), augmented with a variable traffic congestion duration. This generates an adequately random setting for our purposes.

(2) *Shortest Path Calculation.* Vehicles will be travelling between arbitrary places. Therefore, the total travel time between each pair needs to be known. This corresponds to the shortest path problem, which is solved using the Floyd-Warshall algorithm [11].

(3) *Shipment Generation.* A list of shipment instances is created. The origin and destination of each shipment are uniformly selected from the warehouses on the map, with the added constraint that the origin must be different from the destination.

(4) *Cargo Item Generation.* Similarly, a list of cargo item instances is created. Each of them is assigned to a uniformly selected shipment and placed at that shipment's origin.

(5) *Vehicle generation.* A list of vehicle instances is created, and each vehicle is placed at a uniformly selected warehouse.

(6) *Vehicle Loading.* Cargo is placed on vehicles. At each warehouse, the shipments are matched to the available vehicles. The shipments and their cargo items are assumed to be in order of importance and therefore treated sequentially. Each cargo item is placed on the next available vehicle, which is selected in a round-robin fashion. If no vehicle is available, it is assumed that the cargo item will be transported at a later time.

(7) *Itinerary Generation.* After each vehicle has been loaded with cargo items, its itinerary can be calculated. The destination of each shipment is examined. The shortest path to this destination is already known. Each place in this path is added to the itinerary, unless it was already part of a previously examined path and has therefore already been added.

(8) *Travel Simulation.* Once all of the vehicles' itineraries are known, their order lists are carried out. A (global) discrete clock is introduced. At every clock tick, vehicles that are en route move one step closer to their next destination. If a vehicle arrives at a place, services for that place are invoked; we will elaborate on these in the next section. The time required for the execution of these services determines the number of clock ticks spent by the vehicle at each place. Once that time has passed, the vehicle moves on to its next destination or, if its order list has been completed, remains at the current place until the next simulation. This chain of events is visualized in Figure 8.

(9) *Termination.* The simulation is completed once all vehicles have finished their order lists and all the corresponding service invocations have been completed.

4.3. Scenario Implementation Details. Having outlined our logistics simulator itself, we can integrate the three scenarios that we selected from Section 4.1 into it and apply our service platform to execute them.

4.3.1. Cargo Validity Verification (Scenario A). In this scenario, we deploy OWL ontologies to illustrate their applicability to the locale-related challenges of the logistics context.

At a warehouse, before a vehicle is loaded with additional cargo, a service must confirm that that particular cargo may indeed be placed on the vehicle. After all, local regulations might prevent the simultaneous transportation of certain goods.

We take the case where items for human consumption cannot be placed on a vehicle that contains animal fodder, due to risk of contamination. This might actually be hard to detect, because it is not always straightforward to tell if a container is in fact loaded with either class of items, due to confusing or incomplete phrasing, highly technical vocabulary or even different languages altogether. As an example, we focus on the latter.

We assume that certain cargo is labeled with the Dutch word "varkensvoer", meaning "pig feed" and that the corresponding OWL class carries the same name. The word "varkensvoer" might not mean anything in an English-speaking country, but through the introduction of an additional ontology containing translations of local concepts, we can assert that "varkensvoer" is in fact an OWL subclass of "animal fodder". The latter in turn has an object property that prohibits it from occurring on a vehicle that also carries human food.

This information is distributed across ontologies which define region-dependent concepts and ontologies that model

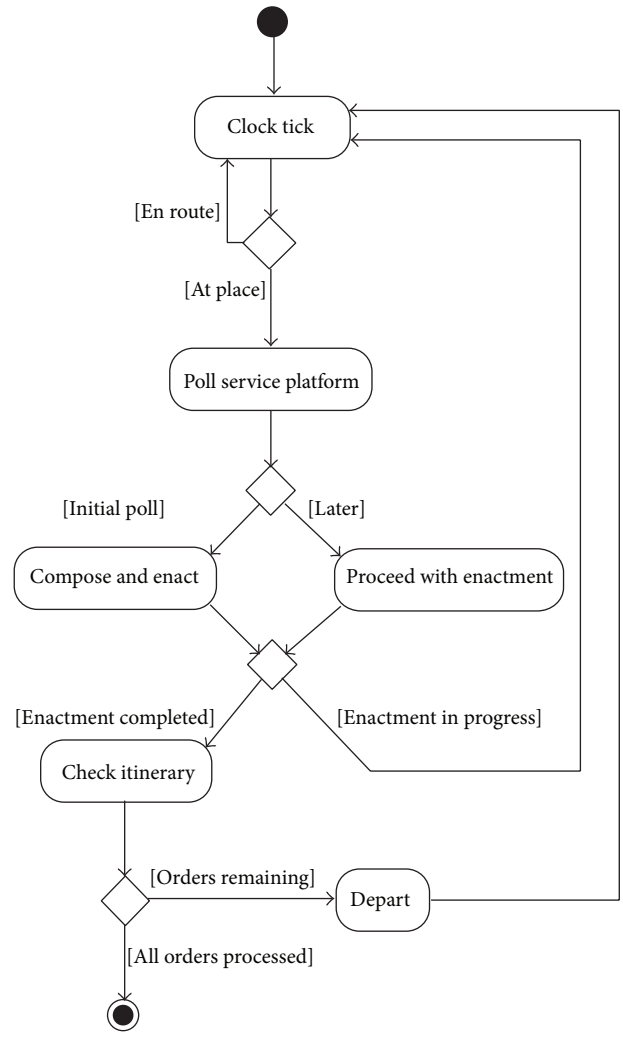


Figure 8: UML activity diagram for actions upon a clock tick in the logistics simulator.

legislation. In our example, there is an ontology that, among other things, asserts that the Dutch-speaking part of Belgium means "pig feed" by "varkensvoer" and a second ontology that expresses a European regulation which prohibits simultaneous transportation of any animal fodder and items for human consumption. The platform itself is already aware of the fact that "pig feed" is a subclass of "animal fodder", allowing its ontological reasoner to infer that "varkensvoer" may not be transported simultaneously with human food. This is summarized in Figure 9.

Of course, this trivial example merely scratches the surface. Thanks to the expressiveness of Semantic Web technology and OWL ontologies in particular, we are confident that they are a highly suitable mechanism for modeling virtually any constraint encountered both in the cargo transportation domain and in the symbiotic networking domain in general.

4.3.2. Sensor Reading Verification (Scenario D). When a vehicle enters a warehouse, all readings from the wireless sensors mounted on the vehicle's cargo items need to be

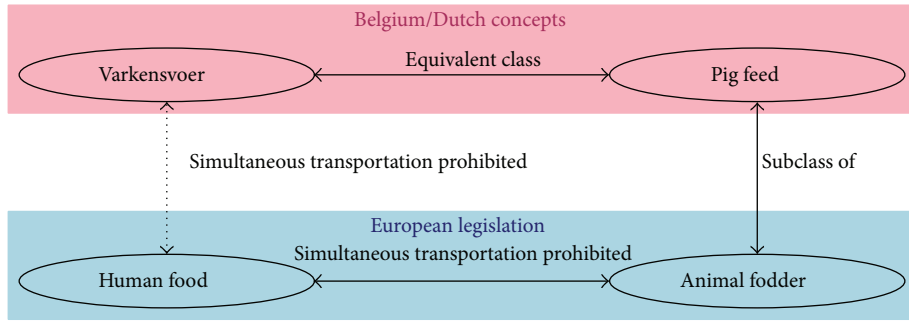


FIGURE 9: Combination of legislation and localization ontologies to infer that "varkensvoer" cannot be transported simultaneously with items for human consumption.

checked. Depending on the type of the cargo, stated in the shipping manifest, the accepted range of each measured property (e.g., temperature, humidity, or pressure) will vary. Moreover, for every property, the sensor node may have been manufactured by a different vendor and thus use a different interface.

We assume that each cargo owner offers services that expose the necessary information manifest of each shipment; the symbiotic network infrastructure accommodates direct communication with those services. Similarly, each sensor node exposes a service for each of its measured properties. One might argue that cargo owners could be reluctant to provide such information; however, we will show that our solution uses it to their advantage.

Furthermore, the warehouse has services available to interpret the output of each sensor vendor's services. This output may, however, require further processing before it can be validated. For instance, a measured temperature may be given in degrees Fahrenheit, whereas validation can only occur on temperatures expressed in degrees Celsius. This is modeled by means of a validation service that takes such a temperature as its sole input.

Depending on the type of the cargo, it may have any or even all of the possible sensor types mounted. Dangerous chemicals might, for instance, be equipped with temperature, humidity, and pressure sensors. Combined with the number of cargo types and the number of sensor device vendors, there is a vast amount of possible service variations. Thus, rather than defining every possible workflow in advance, we construct compositions on demand to attain the goal of sensor measurement validation.

Furthermore, the optimal composition to reach a goal will change over time. The temperature conversion service described above might be offered by various partner networks, so as to implement load balancing. By describing these services in separate ontologies, we can include them in the knowledge base only if they are actually available to the platform.

In our simulator, each arrival of a vehicle at a warehouse triggers the evaluation of a set of SWRL rules against the problem context. For this particular scenario, we use the following rule:

$$CargoItem(?i) \wedge Sensor(?s) \wedge hasSensor(?i, ?s) \Rightarrow VerifiableCargoItem(?i)$$

Thus, if a cargo item is equipped with one or more sensor nodes, then that item is part of the set of items that will be verified. We consider three types of sensors: temperature, humidity, and pressure, each occurring with a predefined probability.

For each cargo item, verification occurs by constructing a composition that checks all of its sensors' outputs. We consider three types of sensors, so the goal service *CargoItemVerifier* takes between one and three inputs. They are of the types *TemperatureVerified, HumidityVerified*, and *PressureVerified*.

These are also the output types of the corresponding services *TemperatureVerifier*, *HumidityVerifier*, and *PressureVerifier*. They each take two inputs. The first input is the sensor reading: *HumidityVerifier* and *PressureVerifier,* respectively, require a generic *Humidity* and *Pressure* individual, whereas *TemperatureVerifier* insists on obtaining a *CelsiusTemperature*. The second input parameter is the shipping manifest.

The shipping manifest can be obtained from a *ShippingManifestProducer* service. The sensor readings are provided by *Producer* services, one per reading type. Temperature sensors use a different temperature scale, depending on the manufacturer of the sensor node. To facilitate conversion from degrees Fahrenheit to degrees Celsius, so the *TemperatureVerifier* service can be deployed, the warehouse network offers the *FahrenheitCelsiusConverter* service.

Figure 10 displays an example of a generated composition, in which the readings from cargo item number 42, equipped with two sensors manufactured by vendor A, are verified. Thus, to construct this composition, the SeCoA algorithm started from the goal service on the right, and gradually satisfied every occurring input parameter until all three initial services were included in the composition.

As described in the previous section, each service included in the composition delays the vehicle by a predefined amount of clock ticks. This amount is determined by the type of the service. Additionally, a discrete cost is associated with each service invocation. Its exact meaning is dependent on the implementation: the *FahrenheitCelsiusConverter*

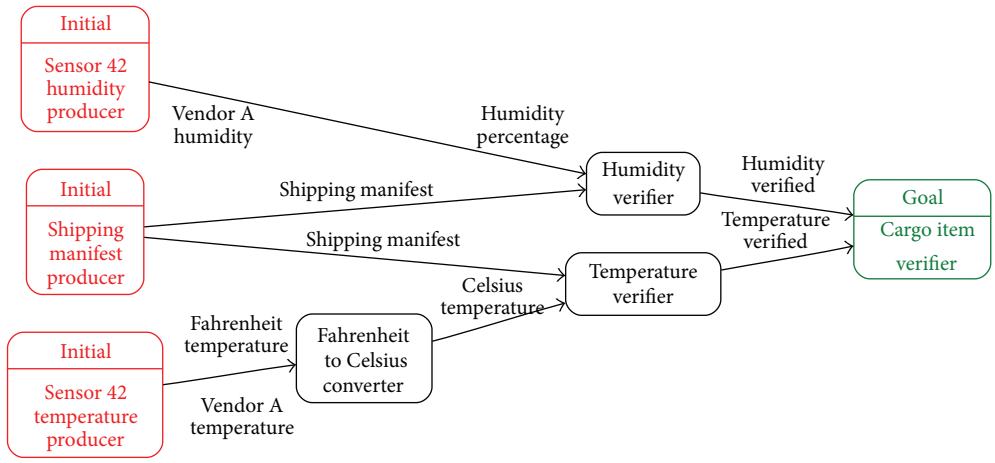


Figure 10: Example composition for sensor reading verification.

might be offered by a third party which charges a fixed fee per invocation, whereas a *HumidityProvider* might increase the energy consumption of a wireless sensor node, thus reducing its lifespan. An analysis of the exact cost of each service is beyond the scope of this paper; therefore, we refrain from further inspection of the constructed compositions' total cost.

4.3.3. Stakeholder Alerting (Scenario E). During the enactment of sensor reading verification, unacceptable readings may have been discovered. In this case, all the stakeholders must be informed of the event, so they can take appropriate action: the vehicle's driver, the personnel of the warehouse, and the owner of the cargo need to be informed, so the cargo can be inspected, destroyed, or perhaps rerouted. The sensor readings might even indicate dangerous conditions, in which case local authorities need to be alerted. In order to do so, we again employ service composition.

Certain parties, namely, the network of the vehicle and that of the warehouse, are assumed to be readily accessible, since they are on site. Local authorities, on the other hand, will not be within walking distance. Moreover, the owner of the cargo might well be halfway around the globe. This does not prevent us from establishing symbiotic relations with their networks, but we do need to take additional precautions to ensure that communication is carried out with the necessary confidentiality. Thus, we will rely on SeCoA's policy mechanism.

The goal service of compositions for alerting is called *StakeholdersAlerter* and takes three inputs: *WarehouseAlerter*, *CargoOwnerAlerter*, and *AuthoritiesAlerter*. Each of these inputs obtains its first input parameter, of the type *Alert*, from the *AlertProducer*; this way, the services know which alert to relay to the stakeholders. Additionally, *CargoOwnerAlerter* and *AuthoritiesAlerter* are instances of the OWL class *RemoteService*, a subclass of OWL-S's generic *Service*. They require an additional input parameter of the type *Tunnel*, which represents a channel for communication with services outside the warehouse realm. Such a *Tunnel* can be obtained from the *TunnelProvider* service.

A *Tunnel* can be augmented with the required confidentiality by securing all communication passing through it. This is achieved by applying the SeCoA filters *Encryption* to the consumer. To state that the warehouse wishes to use encryption, it enforces the following SeCoA policy:

$$sameAs(TunnelProvider, Producer) \wedge RemoteService(Consumer) \Rightarrow \text{apply } Encryption \text{ to } Consumer$$

As outlined in Section 3, the policy's antecedent is a SWRL expression; it checks whether the service producing the parameter is *TunnelProvider* and the one consuming it is a *RemoteService*. Thus, any remote service using a tunnel will be instructed to encrypt its communication.

In Figure 11, a composition to realize such an alerting process is visualized. Note the application of the SeCoA filter *Encryption* to both remote services.

4.4. Evaluation Approach. Our simulator, as well as the symbiotic service platform, and the SeCoA algorithm in particular, were implemented in *Java 7*. For the simulator, we used the *OWL API* [12], while the SeCoA algorithm relies on the *OWL-S API* [13]. Both employ the *Pellet* reasoner [14].

As in our problem context, we used a fictitious square-shaped map, measuring 5,000 by 5,000 distance units; as described in Section 4.2, these determine the travel times between places on the map. To compute the travel times, we augmented the Euclidean distances with uniformly generated traffic congestion durations between 100 and 1,000 clock ticks. In the generated scenarios, up to 200 vehicles transport a total of 500 cargo items between warehouses. Each cargo item belongs to a uniformly selected shipment from a list of 100. Vehicles may contain up to 20 cargo items each. The map contains 50 such warehouses, as well as 10 customs offices, representing country borders.

Each cargo item is equipped with up to three sensors, one of each available type. All items have a temperature sensor; humidity sensors occur with a probability of 50%; pressure sensors occur with a probability of 33%. Each sensor is manufactured by one of three fictitious sensor vendors, which is uniformly selected; two of these vendors use the

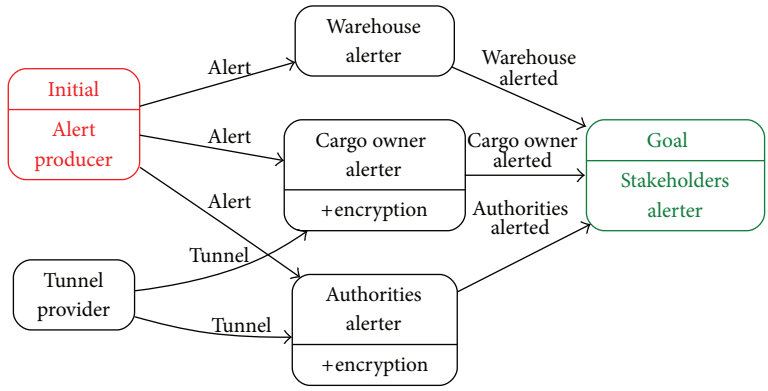


Figure 11: Example composition for sensor reading alerting.

Table 5: Delay parameters for simulated services.

Service	Delay
TemperatureProvider, *HumidityProvider*, and *PressureProducer*	1 tick
TemperatureVerifier, *HumidityVerifier*, and *PressureVerifier*	1 tick
FahrenheitCelsiusConverter	1 tick
TunnelProvider	5 ticks

Fahrenheit temperature scale, while the other uses Celsius, like all of the warehouses' services.

As outlined, whenever a software service is included in a composition, this imposes a delay on the vehicle being examined, expressed as a discrete amount of clock ticks. The delay parameters we used in our simulations are summarized in Table 5; any services that are not listed in this table are assumed to have a negligible delay.

Finally, we assume that the alerting service is required in 20% of all cases. In other words, whenever a vehicle is inspected, there is an 80% chance that all of the cargo items that vehicle is carrying pass verification.

Regarding SeCoA's tunable parameters, we chose α, β, and γ such that the number of initial services not included in the composition would always take precedence over the number of unresolved input parameters, and the latter would always take precedence over the number of services in the composition. This was the most appropriate configuration for our scenarios; other problems or problem domains may require entirely different parameters.

For our simulations, we used two types of machines, so as to emulate the hardware available in real-life transportation and logistics scenarios. Their specifications are shown in Table 6. In particular, the mini PC is an excellent candidate for mass deployment in trucks and warehouses, due to its small size and energy footprint.

Five instances of the described simulation were executed, each using a different random generator seed. The simulations themselves were only carried out on the more powerful desktop PC, as we are less interested in the simulator's raw performance than in the service composition problems originating from it.

The produced composition problems were all tackled by both machines. Thus, each time a vehicle enters a warehouse, a service composition is constructed for the verification of each of the cargo items loaded onto it. As a vehicle usually carries many cargo items, our platform is confronted with large amounts of composition problems. Because these problems are also reasonably small, we experimented with parallel processing. Specifically, each simulation was repeated using 1, 2, 4, and 8 simultaneous composition processes.

We based the maximum of 8 simultaneous processes on the fact that larger amounts would occasionally render the mini PC unresponsive, depending on the complexity of the composition. In the rare event that the composition process would still fail, it was automatically restarted.

4.5. Results and Discussion. The five scenarios that were generated consisted each of between 1,006 and 1,255 composition problems for the purpose of cargo verification and between 21 and 31 for stakeholder alerting. The former are a demonstration of the graph construction aspect of the SeCoA algorithm, while the latter emphasize the subsequent policy processing phase.

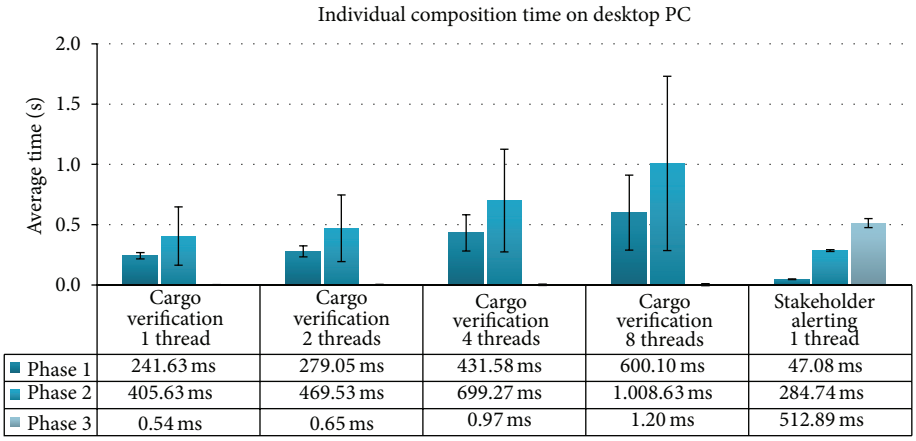
SeCoA required an average of 8.87 iterations to produce a suitable service composition for cargo verification, with a worst-case requirement of 20 iterations. The goal of stakeholder alerting was consistently reached after mere 8 iterations.

In Figures 12 and 13, we show the average execution times for the three phases of the algorithm on both of our evaluation machines. Comparing cargo verification—specifically, the leftmost column—to stakeholder alerting, the impact of policy processing is clearly noticeable. The evaluation of a policy antecedent—that is, a SWRL expression—against each graph edge is quite a resource-intensive process, but is in our opinion a necessary feature of symbiotic service composition. Moreover, the SWRL language provides us with a great amount of flexibility which we would not be able to obtain from a nonsemantic approach.

Two additional differences between the composition types can be observed. Firstly, the first phase of the SeCoA

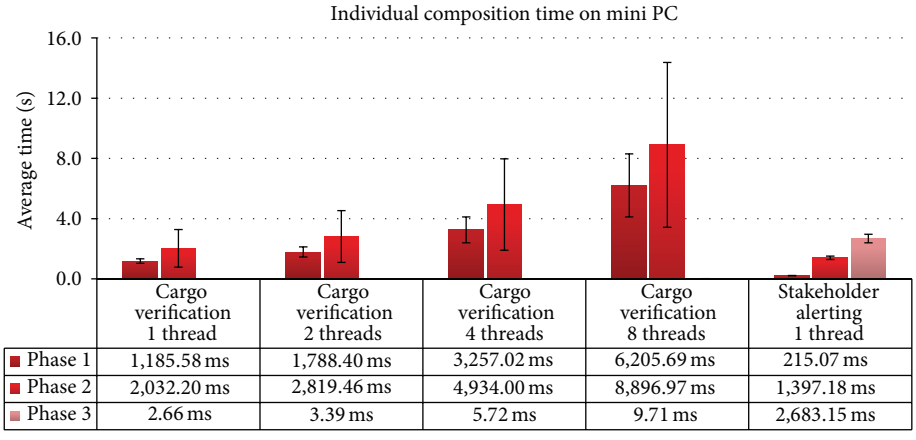
Table 6: Specifications of evaluation hardware.

Type	Software	Processor	Memory
Zotac ZBOX SD ID-10 mini PC	Debian Linux 6.0, 32-bit Java 7	Intel Atom D510 1.66 GHz	3 GB DDR2 800 MHz
Desktop PC	Windows 7 Enterprise, 64-bit Java 7	AMD Phenom II X4 955 3.2 GHz	12 GB DDR3 1.33 GHz



	Cargo verification 1 thread	Cargo verification 2 threads	Cargo verification 4 threads	Cargo verification 8 threads	Stakeholder alerting 1 thread
Phase 1	241.63 ms	279.05 ms	431.58 ms	600.10 ms	47.08 ms
Phase 2	405.63 ms	469.53 ms	699.27 ms	1.008.63 ms	284.74 ms
Phase 3	0.54 ms	0.65 ms	0.97 ms	1.20 ms	512.89 ms

Figure 12: Average amount of time required for both scenario types on the described desktop PC, with different parallelization strategies.



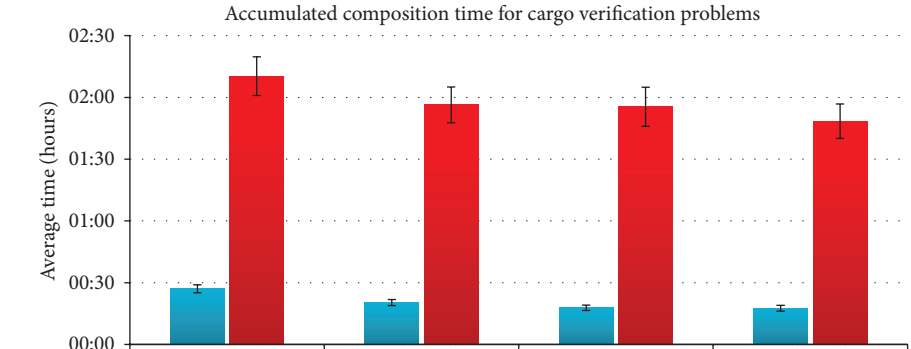
	Cargo verification 1 thread	Cargo verification 2 threads	Cargo verification 4 threads	Cargo verification 8 threads	Stakeholder alerting 1 thread
Phase 1	1,185.58 ms	1,788.40 ms	3,257.02 ms	6,205.69 ms	215.07 ms
Phase 2	2,032.20 ms	2,819.46 ms	4,934.00 ms	8,896.97 ms	1,397.18 ms
Phase 3	2.66 ms	3.39 ms	5.72 ms	9.71 ms	2,683.15 ms

Figure 13: Average amount of time required for both scenario types on the described mini PC, with different parallelization strategies.

algorithm, in which services whose input parameters are not provided anywhere, takes quite a bit longer in the case of cargo verification. This is simply because more services need to be examined in the case of cargo verification: prior to composition, the platform loads semantic descriptions of the sensor data processing services provided by the fictitious manufacturers which we consider. Secondly, the large standard deviations are a consequence of the fact that, in the case of cargo verification, the simulated service ontologies are quite diverse. Ontological reasoning as well as the composition process is greatly affected by such variations. Due to the nature of semantic models, it is usually rather challenging to assess the complexity of a reasoning-supported problem.

Even if the policy application phase takes a bit of time, our algorithm produces the desired composition in just over three seconds. Our objective was to produce compositions that would eliminate manual intervention and greatly reduce the time required for the cargo verification process, and we feel that our solution does just that. Even on our mini PC's moderate hardware, compositions are available after just a few seconds. Also note that, as far as our simulations are concerned, the most time-consuming type of composition process occurs far less frequently, as the stakeholders are only alerted in case of a verification failure. In real-life scenarios, one might of course choose to inform stakeholders more often—a strategy which our platform also supports.

Also the composition times required when we introduce parallelism are shown in Figures 12 and 13. We only do so for the verification process. After all, stakeholder alerting happens on a per-truck basis, so the probability that multiple alerting compositions need to be enacted is rather low. Therefore, parallelization would not affect performance in this case.



	1 thread	2 threads	4 threads	8 threads
Desktop PC	00:27:02	00:20:21	00:17:49	00:17:38
Mini PC	02:10:20	01:56:25	01:55:27	01:48:25

FIGURE 14: Accumulated composition time required for all cargo verification problems in a simulation, with different parallelization strategies and on both evaluation machines.

Observe that the average time required to construct a composition increases with the amount of simultaneous compositions being constructed. This should not come as a surprise, as multiple composition processes need to share system resources. However, the overall composition process, that is, the construction of compositions to verify an entire truck's contents and the total of all these composition sets—does benefit from parallel processing. The degree to which parallel composition increases efficiency is shown in Figure 14. By scaling from a single process to 8 simultaneous ones, the total time spent constructing service compositions is reduced by 20 percent on the mini PC and up to 58 percent on the desktop PC.

Let us also take a brief look at our platform's memory consumption. The service composition process, including its full ontological model, required a maximum of 86 MB of RAM on our mini PC, not including the Java virtual machine; the number of threads used did not affect this upper bound. On the desktop PC, this amount rose to 232 MB. This has little to do with our implementation itself: we deliberately opted for a 64-bit version of Java on the more powerful machine, so as to take full advantage of its resources. 64-bit versions of the Java virtual machine are known to require substantially more memory.

Nevertheless, we still consider 232 MB a low memory footprint. Multiplied by the maximum of 8 simultaneous composition processes, this definitely leaves ample room for further scaling if desired. We however also note that, on the desktop PC, scaling from 4 to 8 simultaneous processes does not yield a tremendous time advantage. Due to the complexity of the overall problem as well as ontological reasoning, the optimal tradeoff between resource allocation and execution time will most likely vary with the problem domain at hand.

5. Scalability Evaluation

5.1. Evaluation Approach. To further assess the performance of our service composition platform and phase 2 of SeCoA in particular, we also developed a generic composition problem

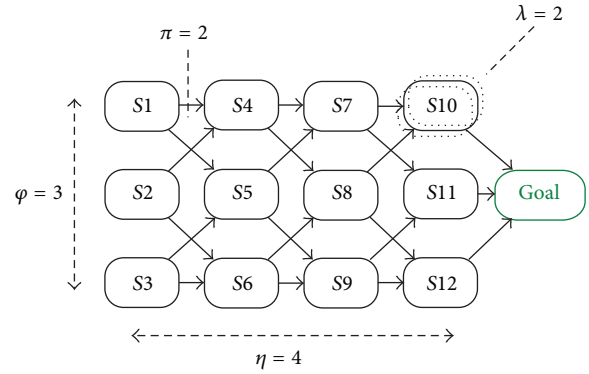


FIGURE 15: Example composition showing problem generator parameters.

generator. The nature of the produced problem is determined by four parameters, also visualized in Figure 15. They are as follows.

(i) Vertical scale φ is the number of services at each "level" of the resulting composition. Because our problem model requires a single goal service, φ is also the number of services which directly deliver output to the goal service.

(ii) Horizontal scale η is the length of a path through the composition, that is, its depth. It is given by the number of parameter exchanges and not the number of services, which is one higher. Thus, the total number of services in the resulting composition is $\varphi \times \eta + 1$, where the last term represents the goal service.

(iii) Alternative service count λ is the number of services which the algorithm must consider before encountering a service suitable for inclusion in the composition. As λ increases, we deliberately make phase 2 of SeCoA consider more services that do not lead to a composition that realizes the goal. We implemented this feature such that an alternative service leads

to a new entry on the priority queue, which is subsequently disposed because providing its input parameters would lead to a cycle in the graph.

(iv) Input parameter count π: is the number of input parameters to each service, apart from the goal service. Each type of input parameter is provided by exactly $1 + \lambda$ services. The first term in this expression is given by the only service which will actually lead to a solution. At each level, the consumed input parameters are assigned to the level producing them in a round-robin fashion. This ensures that the resulting composition always contains all the necessary services.

Let us compare these parameters to those used in our complexity analysis from Section 3.5. We see that π (almost) corresponds to j, the average number of input parameters per service. In reality, π is slightly below j, as j also takes into account the goal service; as the problem scale increases, the relative difference between the two becomes negligible. Furthermore, λ determines the value of both t, the average number of services that provide a parameter type, and s, the total number of services considered. If we were to omit the goal service, t would be equal to $1 + \lambda$, while it is now slightly lower; again, a larger problem scale will relieve this shortcoming. For the total number of services s, the expression is $\varphi \times \eta \times (1 + \lambda) + 1$; the final term represents the goal service. Because we refrain from reusing service output and no policies are applied, $P(R)$ and p are both zero. Our worst-case time complexity expression from Section 3.5, $O((j + p) \times s + (j \times t \times (1 - P(R)))^{j \times s})$, therefore becomes $O(j \times s + (j \times t)^{j \times s})$. We refrain from introducing the newly obtained expressions for j, s, and t. Suffice it to say, because all four of our parameters occur (among others) in the exponent $j \times s$, increasing the value of any of the parameters will make the problem exponentially more complex in the worst case.

The generated problems were solved by the same hardware used in our approach based on simulation of transportation and logistics, shown in Table 6. Again, we employed *Java 7*, the *OWL-S API* [13], and the *Pellet* reasoner [14].

We chose the problem generator parameters φ, η, λ, and π such that we could investigate both the performance of our algorithm and the limitations of our hardware. The number of input parameters π was kept constant, so as not to overcomplicate the scenarios; we consider an average of 2 input parameters per service typical and thus opted for this value. The parameters that determine the dimensions of the composition graph, φ and η, ranged between 2 and 5. Finally, we considered between 0 and 3 alternatives to each service. Thus, in each composition problem, each service had the same number of alternatives. Recall that these alternatives deliberately send our algorithm off on a tangent. Because of the exponential character of the search space, even a relatively low number of alternatives per service allows us to quickly devise a complex problem and consider compositions with many false positives. In real-life scenarios, however, an appropriately chosen evaluation function $p(c)$ will avoid this behavior.

5.2. Results and Discussion. For both the desktop PC and the mini PC, the times required to solve the generated composition problems are shown in Figure 16. One can indeed observe that a higher number of alternative services, combined with an unsuitable evaluation function $p(c)$, causes a sharp rise in execution time. As we also set out to determine the limitations of our hardware, the most complex problems did not yield a solution. In these cases, the desktop PC ran out of memory. We could add more RAMs to solve slightly more challenging problems. However, we can extrapolate from the execution times that composition would still require quite a long time. On the mini PC, we opted for 32-bit Java, which means that the 3 GB of RAM installed is already the memory limit.

On the other hand, like in our logistics simulations, memory usage was lower on the mini PC. This modest machine required just 85 MB of RAM to solve problems, where $\lambda = 0$. For $\lambda = 1$, we saw an average usage of 175 MB, with a maximum of 897 MB when $\varphi = \eta = 5$. If we increase λ beyond 1, we can only consider partial results, as the most complex problems could not be solved. For $\lambda = 2$, we do not have full results for $\varphi = \eta = 5$, so we only note that composition required a maximum of 1.68 GB of RAM for $\varphi = \eta = 4$. If we increase λ to 3, the result set is not adequately complete to formulate further conclusions.

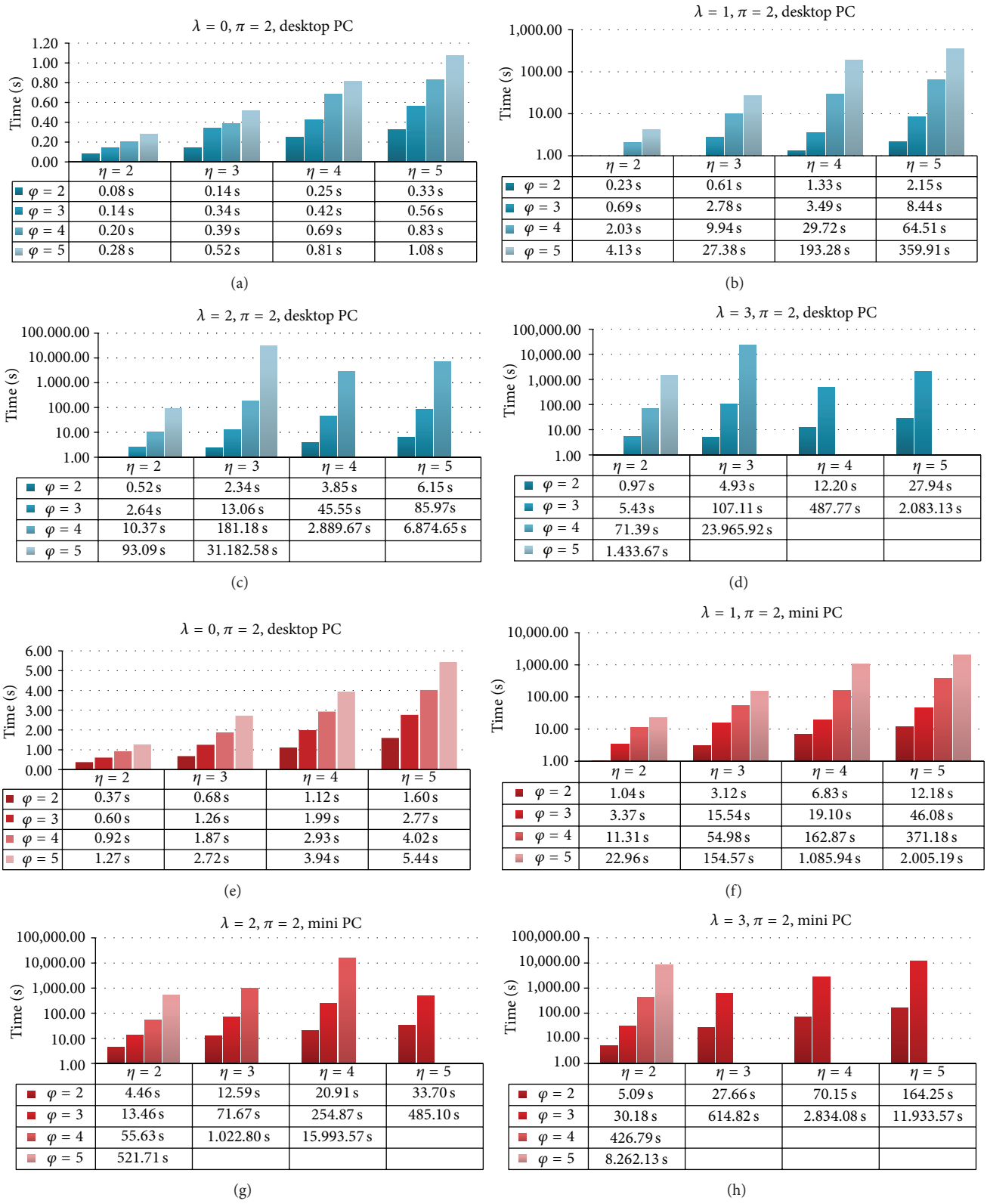
Memory usage on the desktop PC was again considerably higher than that on the mini PC, due to the use of 64-bit Java. Of course, in this case, the much higher memory limit does allow us to solve more complex problems, given adequate time. For $\lambda = 0$, composition on the desktop PC used on average 3.1 times more RAM than on the mini PC. Incrementing λ, and thus the problem scale, increases this factor as well. For $\lambda = 1$, average memory usage on the desktop PC was 4.5 times as high as on the mini PC, an increase by nearly 50%. Because of the incomplete results for $\lambda > 1$, we can only note that the factor luckily does seem to increase more slowly as the problem scales.

Finally, we note that the time and memory used for the full composition process greatly depend on the ontologies at hand and the reasoning performed upon them. In these benchmarks, we opted to keep the impact of Semantic Web technology to a minimum in order to focus on our algorithm itself and specifically its second phase.

6. Related Work

Software service composition, semantic or otherwise, has seen a lot of research interest. An important advantage of the OWL-S ontology over other formalisms, such as BPEL, is that it allows for expressing nonfunctional properties such as quality of service [15]. Such nonfunctional properties are crucial to the context of symbiotic networks, as they introduce many additional constraints. OWL-S itself is rather limited in this area, but relying on OWL, one can extend the ontology with the necessary concepts, as we have also demonstrated.

Several tools that exist facilitate the service composition process. The semiautomatic OWL-S service composer by



	$\eta = 2$	$\eta = 3$	$\eta = 4$	$\eta = 5$
$\varphi = 2$	0.08 s	0.14 s	0.25 s	0.33 s
$\varphi = 3$	0.14 s	0.34 s	0.42 s	0.56 s
$\varphi = 4$	0.20 s	0.39 s	0.69 s	0.83 s
$\varphi = 5$	0.28 s	0.52 s	0.81 s	1.08 s

	$\eta = 2$	$\eta = 3$	$\eta = 4$	$\eta = 5$
$\varphi = 2$	0.23 s	0.61 s	1.33 s	2.15 s
$\varphi = 3$	0.69 s	2.78 s	3.49 s	8.44 s
$\varphi = 4$	2.03 s	9.94 s	29.72 s	64.51 s
$\varphi = 5$	4.13 s	27.38 s	193.28 s	359.91 s

	$\eta = 2$	$\eta = 3$	$\eta = 4$	$\eta = 5$
$\varphi = 2$	0.52 s	2.34 s	3.85 s	6.15 s
$\varphi = 3$	2.64 s	13.06 s	45.55 s	85.97s
$\varphi = 4$	10.37 s	181.18 s	2.889.67 s	6.874.65 s
$\varphi = 5$	93.09 s	31.182.58 s		

	$\eta = 2$	$\eta = 3$	$\eta = 4$	$\eta = 5$
$\varphi = 2$	0.97 s	4.93 s	12.20 s	27.94 s
$\varphi = 3$	5.43 s	107.11 s	487.77 s	2.083.13 s
$\varphi = 4$	71.39 s	23.965.92 s		
$\varphi = 5$	1.433.67 s			

	$\eta = 2$	$\eta = 3$	$\eta = 4$	$\eta = 5$
$\varphi = 2$	0.37 s	0.68 s	1.12 s	1.60 s
$\varphi = 3$	0.60 s	1.26 s	1.99 s	2.77 s
$\varphi = 4$	0.92 s	1.87 s	2.93 s	4.02 s
$\varphi = 5$	1.27 s	2.72 s	3.94 s	5.44 s

	$\eta = 2$	$\eta = 3$	$\eta = 4$	$\eta = 5$
$\varphi = 2$	1.04 s	3.12 s	6.83 s	12.18 s
$\varphi = 3$	3.37 s	15.54 s	19.10 s	46.08 s
$\varphi = 4$	11.31 s	54.98 s	162.87 s	371.18 s
$\varphi = 5$	22.96 s	154.57 s	1.085.94 s	2.005.19 s

	$\eta = 2$	$\eta = 3$	$\eta = 4$	$\eta = 5$
$\varphi = 2$	4.46 s	12.59 s	20.91 s	33.70 s
$\varphi = 3$	13.46 s	71.67 s	254.87 s	485.10 s
$\varphi = 4$	55.63 s	1.022.80 s	15.993.57 s	
$\varphi = 5$	521.71 s			

	$\eta = 2$	$\eta = 3$	$\eta = 4$	$\eta = 5$
$\varphi = 2$	5.09 s	27.66 s	70.15 s	164.25 s
$\varphi = 3$	30.18 s	614.82 s	2.834.08 s	11.933.57 s
$\varphi = 4$	426.79 s			
$\varphi = 5$	8.262.13 s			

FIGURE 16: Amount of time required for service composition on the described hardware. Logarithmic scale used where $\lambda > 0$.

Sirin et al. guides the user in mostly manual composition construction [16]. Using logic-based reasoning and text similarity measurement, the hybrid *OWLS-MX Matchmaker* attempts to automatically select the most suitable service [17].

Hierarchical task network (HTN) planning has been used as a mechanism for service composition by several researchers. The intermediate translation to PDDL seen in *OWLS-XPlan* [18] could raise performance concerns in resource-constrained environments such as symbiotic networks. Another HTN-based composer is *SHOP2* [19], which uses a proprietary model. The *WTE+* project focused on building mashups through QoS-aware HTN planning [7].

SeCoA differs from such efforts in that it takes into account additional, nonfunctional constraints, related to the symbiotic nature of the network environments involved. Additionally, it introduces a tunable priority function, which better accustoms the composition logic to the nature of the environment.

Metaheuristics are another common approach toward service composition. In [20], a variation of genetic algorithms is applied to a similar problem. However, the model builds upon traditional Web services rather than the Semantic Web, sacrificing a degree of expressiveness.

The service platform that we introduced does not yet attempt to evaluate policies incrementally. Bahati and Bauer analyze the modifications policies may undergo and implement an adaptive approach [21]. Conflict resolution in ontology-based policies is discussed by Barron et al. [22].

The complexity analysis of our algorithm does not take into account the ontological reasoning that precedes it. In our simulations, we used the Pellet reasoner. Bock et al. benchmark several reasoners and conclude that none of them is superior in all areas [23]. Kang et al. propose a model to predict reasoning performance based on ontology metrics [24].

Kim et al. argue that providing ontological descriptions of services and their contexts is a task not to be taken lightly and provide a context-aware modeling technique [25]. Such a technique could be beneficial to the actors in the transportation and logistics scenarios which we emulate.

To assess a system's capabilities in terms of ontological reasoning, a few standardized ontologies exist. The *University Ontology Benchmark* provides OWL ontologies of varying sizes [26]. One can also use the *OWL-S Service Retrieval Test Collection (OWLS-TC)* to evaluate tools that make use of service ontologies [27]. Our simulator does not employ these generic benchmarks, as its ontologies are heavily tailored to the logistics domain.

In our simulator, we deal with relatively small ontologies and service compositions. As the problem scale increases, we might no longer be able to parallelize the process on a single machine. Conversely, one could explore the possibility of more resource-constrained hardware. Verstichel et al. present an approach toward distribution of ontological reasoning tasks [28].

Countless simulators exist to create a highly realistic model of road traffic and cargo transportation. *FreeSim* simulates freeway systems at the macroscopic and the microscopic scale [29]. A decision-based multiagent simulation of transport chains can be achieved using *TAPAS* [30]. Caris et al. simulate container barge traffic in the port of Antwerp [31]. However, we view logistics simulation as more of a means than an end. As described in Section 4, we opted for the implementation of a simplified simulator, which underpins the service composition platform that we presented in this paper.

7. Conclusions and Future Work

We introduced symbiotic networks—network environments that intelligently share resources and autonomously adapt to the dynamicity thereof—and proposed a service platform for the autonomous operation of software services in such networks. Building upon Semantic Web technology, we are able to describe such services in an expressive fashion and formalize the policies that govern the interaction of symbiotically interoperating partners. Our tunable best-first search algorithm SeCoA produces service compositions that realize semantically described goals.

We applied our service model and the SeCoA algorithm to scenarios from the field of cargo transportation and logistics, where sensor network readings are processed, verified, and reacted upon. We showed that the Semantic Web-based approach is highly suited for modeling concepts specific to this domain, even in an international context with ample concepts that are often defined ambiguously. Through the implementation of a logistics simulator, we created a context in which semantically defined goals arise, which our service platform subsequently addresses.

Performance evaluations showed that our approach delivers the required service compositions within a matter of seconds, even on resource-constrained devices like the ones typically seen in the cargo transportation and logistics domain, among others. Through parallelization of composition processes, we are able to provide a scalable platform that meets the needs of such a context. We are therefore confident that in real-life scenarios symbiotic networking and symbiotic services in particular can streamline day-to-day processes and increase overall efficiency.

Additionally, we used a generic problem generator to investigate the scalability of our algorithm. We showed that its time and memory requirements are acceptable for smaller problems. Gradually increasing the problem scale revealed that the requirements do tend to grow rapidly. However, SeCoA accommodates to this by depending on a tunable priority function, which determines the path through the search space.

As a next step, we will be researching the enactment of the logistics scenarios we have simulated, as well as other service compositions from this and other fields. To achieve policy-aware composition enactment in symbiotically enabled wireless sensor networks, we are currently in the process of integrating the *CaPI* middleware system [32].

To allow for more expressive service modeling and composition, the SeCoA algorithm will be extended with the remaining control constructs from the OWL-S ontology. We will also look into alternative algorithms and heuristics for service composition.

Finally, as mentioned, Semantic Web technology is merely one approach toward highly expressive service modeling. Rather than confronting system administrators and end users with the at times daunting formalisms of OWL, OWL-S, and SWRL, we aim for a highly intuitive and accessible user interface.

Acknowledgments

Tim De Pauw wishes to thank the University College Ghent Research Fund for financial support through his Ph.D. grant. Part of this work has been funded by the IWT SBO SymbioNets project.

References

[1] E. de Poorter, B. Latré, I. Moerman, and P. Demeester, "Symbiotic networks: towards a new level of cooperation between wireless networks," *Wireless Personal Communications*, vol. 45, no. 4, pp. 479–495, 2008.

[2] E. de Poorter, P. Becue, M. Rovcanin, I. Moerman, and P. Demeester, "A negotiation-based networking methodology to enable cooperation across heterogeneous co-located networks," *Ad Hoc Networks*, vol. 10, no. 6, pp. 901–917, 2012.

[3] B. Latré, B. Braem, I. Moerman, C. Blondia, and P. Demeester, "A survey on wireless body area networks," *Wireless Networks*, vol. 17, no. 1, pp. 1–18, 2011.

[4] D. Martin, M. Burstein, J. Hobbs et al., "OWL-S: semantic markup for web services," W3C Member Submission, 2004, http://www.w3.org/Submission/OWL-S/.

[5] D. L. McGuinness and F. van Harmelen, "OWL Web Ontology Language overview," 2004, http://www.w3.org/TR/owl-features/.

[6] I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: a Semantic Web Rule Language combining OWL and RuleML," W3C Member Submission, 2004, http://www.w3.org/Submission/SWRL/.

[7] A. Hristoskova, B. Volckaert, and F. de Turck, "The WTE+ framework: automated construction and runtime adaptation of service mashups," *Automated Software Engineering*, vol. 20, no. 4, pp. 499–542, 2013.

[8] T. de Pauw, B. Volckaert, V. Ongenae, and F. de Turck, "SeCoA: autonomous semantic service composition algorithm in symbiotic networks," in *Proceedings of the 13th IEEE/IFIP Network Operations and Management Symposium (NOMS '12)*, Maui, Hawaii, USA, 2012.

[9] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Reading, Mass, USA, 1969.

[10] Stanford Center for Biomedical Informatics Research, "The protégé ontology editor and knowledge acquisition system," http://protege.stanford.edu/.

[11] R. W. Floyd, "Algorithm 97," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[12] University of Manchester, "The OWL API," http://owlapi.sourceforge.net/.

[13] T. Möller, "OWL-S API," http://on.cs.unibas.ch/owls-api/.

[14] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: a practical OWL-DL reasoner," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 51–53, 2007.

[15] N. Milanovic and M. Malek, "Current solutions for Web service composition," *IEEE Internet Computing*, vol. 8, no. 6, pp. 51–59, 2004.

[16] E. Sirin, B. Parsia, and J. Hendler, "Filtering and selecting semantic web services with interactive composition techniques," *IEEE Intelligent Systems*, vol. 19, no. 4, pp. 42–49, 2004.

[17] M. Klusch, B. Fries, and K. Sycara, "Automated semantic web service discovery with OWLS-MX," in *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*, pp. 915–922, Hakodate, Japan, May 2006.

[18] M. Klusch, A. Gerber, and M. Schmidt, "Semantic Web service composition planning with OWLS-Xplan," in *Proceedings of the AAAI Fall Symposium*, pp. 55–62, Arlington, Va, USA, November 2005.

[19] E. Sirin, B. Parsia, D. Wu, J. Handler, and D. Nau, "HTN planning for web service composition using SHOP2," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 4, pp. 377–396, 2004.

[20] S. Dong and W. Dong, "A qoS driven web service composition method based on ESGA (Elitist Selection Genetic Algorithm) with an improved initial population selection strategy," *International Journal of Distributed Sensor Networks*, vol. 5, no. 1, pp. 54–54, 2009.

[21] R. M. Bahati and M. A. Bauer, "Towards adaptive policy-based management," in *Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium (NOMS '10)*, pp. 511–518, Osaka, Japan, April 2010.

[22] J. Barron, S. Davy, and B. Jennings, "Conflict analysis during authoring of management policies for federations," in *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM '11)*, pp. 1180–1187, Dublin, Ireland, May 2011.

[23] J. Bock, P. Haase, Q. Ji, and R. Volz, "Benchmarking OWL reasoners," in *Proceedings of the Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (ARea '08)*, Tenerife, Spain, 2008.

[24] Y. B. Kang, Y. F. Li, and S. Krishnaswamy, "Predicting reasoning performance using ontology metrics," in *The Semantic Web—ISWC 2012*, vol. 7649 of *Lecture Notes in Computer Science*, pp. 198–214, Springer, 2012.

[25] J. D. Kim, J. Son, and D. K. Baik, "CA_{5W1H}Onto: ontological context-aware model based on 5W1H," *International Journal of Distributed Sensor Networks*, vol. 2012, Article ID 247346, 11 pages, 2012.

[26] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu, "Towards a complete OWL ontology benchmark," *The Semantic Web: Research and Applications*, vol. 4011, pp. 125–139, 2006.

[27] M. Klusch, P. Kapahnke, B. Fries, M. A. Khalid, and M. Vasileski, "OWL-S service retrieval test collection," 2010, http://semwebcentral.org/projects/owls-tc/.

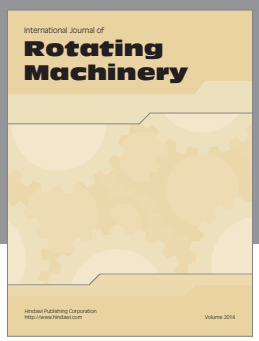
[28] S. Verstichel, B. Volckaert, B. Dhoedt, P. Demeester, and F. de Turck, "Context-aware scheduling of distributed DL-reasoning tasks in wireless sensor networks," *International Journal of Distributed Sensor Networks*, vol. 2011, Article ID 521810, 24 pages, 2011.

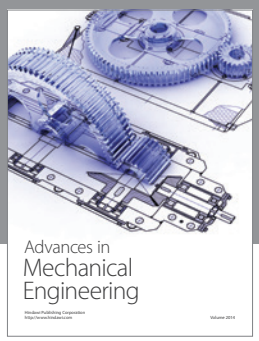
[29] J. Miller and E. Horowitz, "FreeSim—a free real-time freeway traffic simulator," in *Proceedings of the IEEE Conference on Intelligent Transportation Systems (ITSC '07)*, pp. 18–23, Seattle, Wash, USA, October 2007.

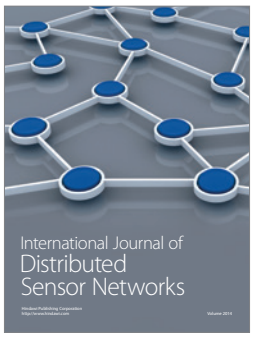
[30] P. Davidsson, J. Holmgren, J. A. Persson, and L. Ramstedt, "Multi agent based simulation of transport chains," in *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '08)*, Richland, SC, USA, 2008.

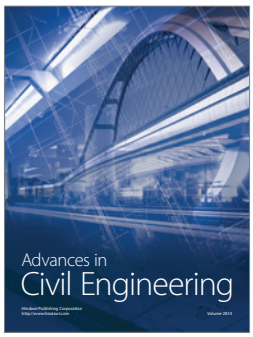
[31] A. Caris, C. Macharis, and G. K. Janssens, "Network analysis of container barge transport in the port of Antwerp by means of simulation," *Journal of Transport Geography*, vol. 19, no. 1, pp. 125–133, 2011.

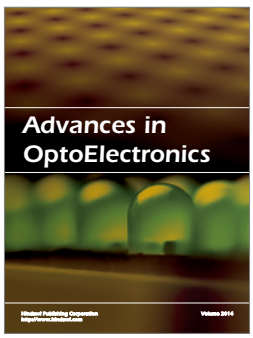
[32] N. Matthys, C. Huygens, D. Hughes, S. Michiels, and W. Joosen, "A component and policy-based approach for efficient sensor network reconfiguration," in *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY '12)*, Chapel Hill, NC, USA, 2012.

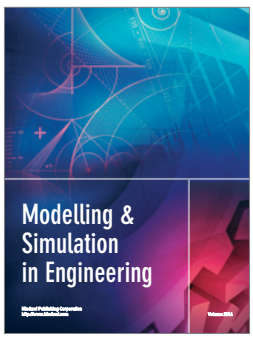


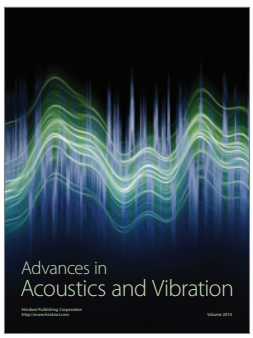


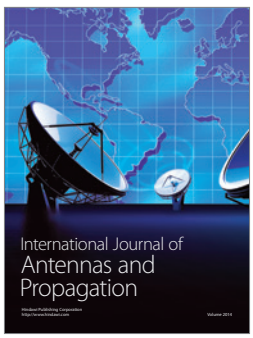


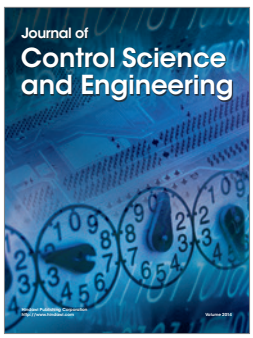


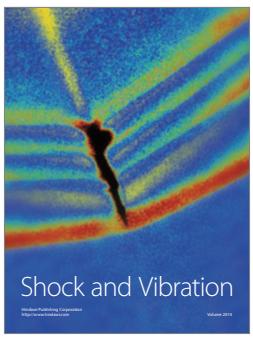












Hindawi

Submit your manuscripts at
http://www.hindawi.com